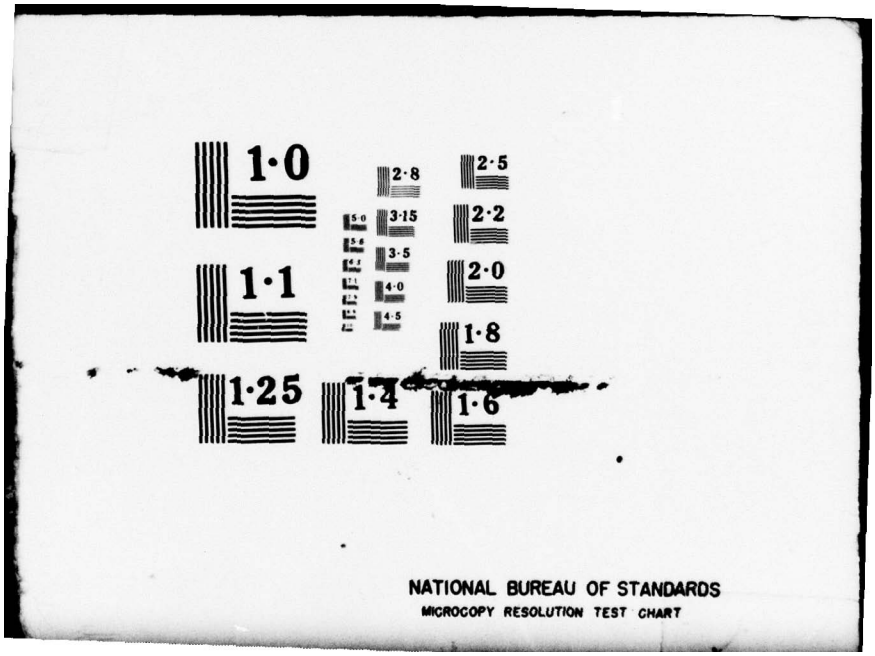
AD-A055 973 NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA F/G 9/2
FUNCTIONAL DESCRIPTION OF A VALIDATION TOOL FOR CMS-2 SOFTWARE.(U)
FEB 78 R N GOSS
UNCLASSIFIED NOSC/TD-138 NL

| OF |
ADA
055973

END
DATE
FILMED
8-78
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

LEVEL II

12 BS.

NOSC

NOSC TD 138

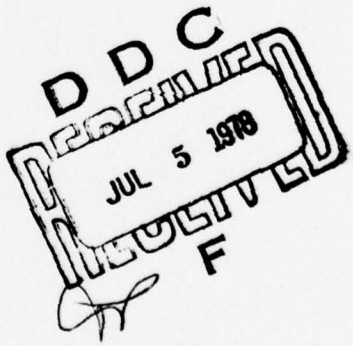
NOSC TD 138

Technical Document 138

FUNCTIONAL DESCRIPTION OF A VALIDATION TOOL FOR CMS-2 SOFTWARE

RN Goss

1 February 1978



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

RR GAVAZZI, CAPT, USN
Commander

HL BLOOD
Technical Director

ADMINISTRATIVE INFORMATION

The work was performed by the Computer Systems Architecture Branch of the Naval Ocean Systems Center (formerly the Naval Electronics Laboratory Center) under project number F61212 (NELC Z291), program element 62766N.

Released by
RR Eyres
Tactical Computer Systems
Architecture Division

Under authority of
JH Maynard, Head
Command Control-
Electronic Warfare
Systems and Technology
Department

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NOSC Technical Document 138 (TD 138) (4 NOSC / TD-138)
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER (9)
4. TITLE (and Subtitle) (6): FUNCTIONAL DESCRIPTION OF A VALIDATION TOOL FOR CMS-2 SOFTWARE
5. TYPE OF REPORT & PERIOD COVERED: Technical document
6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): RN Goss (10) R. N. Goss
8. CONTRACT OR GRANT NUMBER(s) (16)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Ocean Systems Center, San Diego, CA 92152
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: F61212 (NELC Z291); 62766N (11)
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Ocean Systems Center, San Diego, CA 92152
12. REPORT DATE: 1 February 1978 (12) 16 p.
13. NUMBER OF PAGES: 14
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
15. SECURITY CLASS. (of this report): UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software engineering
Software validation
CMS-2 programming language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document sets forth the requirements for a set of computer programs yielding information about tests made on CMS-2 software. This set of programs is called a validation tool. A validation tool, depending on the needs of the investigator, can be made as sophisticated as required at the expense of additional memory and running time. The validation tool proposed here is primarily for static analysis.

393 159

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CONTENTS

1.0 GENERAL

1.1 PURPOSE OF FUNCTIONAL DESCRIPTION

This functional description is written to provide requirements for a set of computer programs yielding information about tests made on CMS-2 software. This set of programs, called a validation tool, is intended to be placed at the disposal of those charged with quality assurance of CMS-2 software planned for inclusion in Fleet tactical combat systems. Since the validation tool is not destined for the Fleet, it does not interface with other system software in the usual sense and is not normally available to the users in the Fleet. Hence, this functional description must be understood in terms appropriate to the intended use of the tool.

1.2 PROJECT REFERENCES

A report, item 2 below, has been written simultaneously with this functional description to amplify the rationale on which it is based. It should be read in conjunction with this document.

1. Project request. "Command Control Distributed System Design and Validation Processors." Research and Technology Work Unit Summary.

2. Naval Ocean Systems Center Technical Document 139, Issues and Perspectives in the Validation of Tactical Software, by RN Goss, 1 February 1978.

3. User's Handbook for AN/UYK-20(V) Computer, vol IV, Change 4, Sperry-Rand Corp, May 1976.

4. User's Reference Manual for Compiler, Monitor System-2 (CMS-2) for Use With AN/UYK-7 Computer, M-5035, vol II, FCDSSA, CA, Nov 1973.

5. CMS-2Q Programmer's Reference Manual, M-5012, revision 5, vol I, II, III, FCDSSA, CA, 1 Sept 1977.

2.0 SYSTEM SUMMARY

2.1 BACKGROUND

The problem addressed in this document is that of ensuring adequate testing of software destined for tactical combat systems in the Fleet. Competent testing is, of course, a major desideratum for such software and needs no rationalizing. Enough has been written on the present unsatisfactory state of software certification to make unnecessary any rehearsal of its shortcomings here.

One of the thorniest questions that the quality assurance investigator must come to terms with is how much testing to do. Ideally, he would like to be able to say that a program which he has certified will operate exactly as it is supposed to under every conceivable circumstance; it is, of course, impossible in nontrivial situations to try out all the alternatives. Such an ambitious undertaking would be tantamount to a formal proof in the logical sense.

The fact that a program is an algorithm, or perhaps a succession of algorithms, has inspired hope in some quarters that rigorous proofs may be a distinct possibility for validating programs, and claims of progress in this direction have been made. Whether formal analysis will be practical for quality assurance of Navy systems has yet to be convincingly argued; in any event, it is years in the future.

In the meantime, we must rely on the only live option we have – the verification of the action of a program through partial testing. If tests in sufficient number and variety are successfully executed, it is felt that the program does, in fact, meet its objectives. This is, out of necessity, a tentative inference. The possibility of errors undetected by the tests already made always exists. The goal must be to minimize that possibility by means of test strategies that efficiently filter the potential sources of problems.

Clearly, if a given program is more than trivial, it is unlikely that tests devised in an armchair environment are going to cover all the contingencies that may arise when a program is executed in the field. Because the mental equipment of the human being is not habituated to threading through exhaustive combinatorial processes, he is not in a position to anticipate what courses an intricate program may take. His tests, more often than not, will leave large areas of the program unexplored and others overtested.

To remedy the foregoing, several efforts are under way to enable the test engineer to plan his work rationally. Validation tools, as the products are called, do not take the place of tests nor do they deal with the content of tests as such. They do furnish information to the investigator about what he has and what he has not tested. Armed with this information, he is able to make his testing more effective and more efficient and with great potential savings in time and cost.

The principle on which these tools are based is that the set of paths which can be traversed in a given program can be identified and dealt with systematically. A program in fact, can be interpreted as a directed graph–the mathematical theory of which has been vigorously developed over the last two decades. It is not a formidable problem to distinguish the arcs of a graph or to insert probes–special instructions for reporting when each arc is passed over–to count the number of traversals. The nodes correspond to branch points.

The primary data are thus execution counts. For a branch we can tell how many times each alternative was carried out and for a loop how many times it was cycled through. From the primary data, summaries and statistics can be derived and on these bases the investigator will decide what additional testing is required. The information will guide him in designing a minimal effective set of tests, and will indicate those parts of the program which carry the heaviest traffic–that is, parts which should be considered as candidates for optimization.

At the expense of additional memory and running time, validation tools can be made more sophisticated. Depending on the needs of the investigator, a tool can be fashioned to check almost any aspect of the program structure and to print the results in the desired format. Certain syntactical information can also be retrieved and used to advantage. Since such tools tend to generate considerable overhead and, as they get more complicated,

to spawn validation problems of their own, it is important that the test engineer demand of the tools only those data that return information of value.

It should be emphasized that, like any other tool, a validation tool is an inert instrument. The results obtained by its use depend upon the skill with which it is applied and are in no sense automatically handed over. Each piece of software to be validated will present to the test engineer a different problem. He has to discover what the problem is and, taking into account the time and cost constraints under which he has to operate, apply his knowledge and resourcefulness to devise tests that exploit the tools to maximum advantage in order to wring from the raw code the facts he needs.

2.2 OBJECTIVES

Validation tool is the name given to the program or group of programs for which requirements are being developed here. The purpose of a validation tool is to return information about tactical computer software written in CMS-2, the language that has been specified for most existing Navy tactical combat systems. Used in conjunction with independently devised tests, the tool will be applicable to any compilable program written in CMS-2M, CMS-2Y, or CMS-2Q language. The nature of the information to be returned will be described fully in the succeeding paragraphs.

The tool will consist of software which can be merged with or applied to any operational CMS-2 compiler to produce an augmented (instrumented) compiler that will run on any machine the corresponding unaugmented compiler will run on, for example, CMS-2Y on the AN/UYK-7 and CMS-2M on the AN/UYK-20. Programs compiled by means of the augmented compiler, called instrumented programs, will retain all the static properties they enjoy when compiled by the unaugmented compiler. The validation tool, which is not necessarily written in CMS-2, will be robust with respect to minor changes in the unaugmented compiler.

2.3 EXISTING METHODS AND PROCEDURES

The reason for interest in validation tools is that no systematic means of generating test data for the CMS-2 software mentioned in paragraph 2.2 are now available to the quality assurance investigator. Present practice consists in the use of one's best judgment to determine whether or not a particular program has been adequately tested. The validation tool does not obviate the exercise of judgment. The problem is that computer programs, especially those involving command and control, are highly combinatorial, and require extensive bookkeeping procedures to keep track of the sequences of decisions that become possible under different aggregates of conditions. Biological evolution has not adapted the ordinary human mind to deal efficiently with such complexity. Consequently, the unaided intellect is a poor resource for making certain that highly ramified decision structures indeed do what they are supposed to do. In the absence of other alternatives, certification of programs has been up to now strictly a human endeavor. The result has been to question whether its very function–to forestall unpleasant surprises when the software becomes operational–has been accomplished.

2.4 PROPOSED METHODS AND PROCEDURES

The most expedient answer to the problem in paragraph 2.3 and the one with which these requirements are concerned, is to complement the unique abilities of the human being for discriminating judgment with those of the machine for tedious record keeping to yield computer-aided test generation. The greatest need of the test engineer, once he has written a series of tests for a program under examination, is to know exactly what has and has not been covered by the tests. What has been overtested, undertested, or untested? With the answers to these questions in hand, he can concentrate his attention where it is needed.

Given a software item to be certified under the present proposal, the quality assurance investigator will devise an initial group of tests based upon his best professional judgment. The tests will be programmed to run with the validation tool. At the conclusion of the run, the investigator will have not only the results of the tests but also data on how the tests have exercised the software under examination printed out according to his directions. The primary data will be execution counts obtained by means of probes inserted in the software. These data will be summarized according to a number of options from which the investigator may choose in order to obtain the best representation of the testing profile of the specific software. He will then be in a position to supplement or revise his first tests. The process will be repeated until he is satisfied that further testing would be redundant or uneconomical. The exact criteria will vary from one situation to another, depending upon the characteristics of the software under analysis.

It is clear that the information provided by the validation tool is not without cost. The probes, which are transfers of control to auditing subroutines, add to the overhead of the uninstrumented program and introduce real-time delays at the points where they have been inserted. These delays may well distort the dynamic properties of the program in cases where timing is crucial. For this reason, the validation tool in the form proposed here is recommended primarily for static analysis, ie, testing of those properties of the program that depend on the formal syntax and structural complexity of the program. The issues arising out of instrumenting programs for real-time analysis are discussed in paragraph 2.4.1.1.

2.4.1 SUMMARY OF IMPROVEMENTS

As already indicated, the benefits which the availability of a validation tool will confer are those resulting from a more efficient test strategy. The test engineer will have a clearer understanding of what tests he needs to design for full coverage of the validation spectrum. The major beneficiary will be the system user who will have in his hands software that will be more reliable and predictable.

It is to be stressed that the instrumentation for execution counts will not be present in the product delivered to the user. The probes are compiled with the user program only for the benefit of the quality assurance investigator. When he is satisfied that the testing is complete, the program is recompiled without the instrumentation software; hence, there is no degradation whatever in the operational program. The only differences possible between the original program submitted for quality assurance and the final program approved for the user would be those improvements suggested by the results of the tests. In such a case, the test engineer would advise the programmer that the program under consideration had failed certain tests; the programmer would then make the needed modification and resubmit

the item to quality assurance for retesting. The user would have no reason to know that this sequence of events had ever taken place.

It is probable, of course, that more thorough testing by quality assurance would mean more extensive revision of a program before it is certified for use; this could result in a delayed delivery date. If such a delay buys a significantly better product, it can by no stretch of the language be reckoned as adverse.

2.4.1.1 DYNAMIC ANALYSIS. Since the real-time properties of a program can be affected by the act of observation, the tester must use care in constructing tests which are not independent of the real-time environment. The elapsed execution time for any program can be obtained from before-and-after readings of the computer's clock. As for clock readings taken in the course of program execution, it is assumed that their normal purpose is to indicate the order in which certain critical events occur (which is not otherwise apparent) rather than the time intervals between events.

2.4.2 SUMMARY OF IMPACTS

The effect of improved test analysis on the software development cycle considered as a process will be minimal. In fact, it will be virtually nonexistent. It adds no new function nor does it relocate any existing function in the lineup. In particular, it does not relieve the programmer of any debugging obligation, because a program is assumed to be in proper condition for use by the time it reaches quality assurance.

2.4.2.1 EQUIPMENT IMPACTS. None.

2.4.2.2 SOFTWARE IMPACTS. No additions to existing application and support software programs are required. The only modification is in the instrumentation of a program for analysis at the time of quality assurance. As indicated above, this modification is temporary; all evidence of the modification is removed before the program is turned over to the user. The validation tool is a piece of software which occupies a modest amount of memory on-line during testing and off-line during the preprocessing and postprocessing phases.

2.4.2.3 ORGANIZATIONAL IMPACTS. None.

2.4.2.4 OPERATIONAL IMPACTS. Since quality assurance takes place before a program is released for operational use, it has no direct effect on the way the program enters into the tactics it is designed to support. Its indirect influence can be considerable. If it does not contribute to a regime in which software troubles are greatly reduced, it will not have accomplished its purpose. The exact leverage it exerts will vary from case to case and can not be made specific, since before-and-after comparisons would not serve any useful purpose.

3.0 DETAILED CHARACTERISTICS

3.1 SPECIFIC PERFORMANCE REQUIREMENTS

The product to be delivered is a validation tool, an instrumentation of computer programs to be used in conjunction with performance tests to yield information about the execution properties of operational software. This tool will permit compilation in and with the source language of the software in question when tests are to be made on or by means of that software. The tool will not normally be present under field conditions. In order to accomplish its purpose, the validation tool must carry out three sequence functions. First, it must ascertain the logical and functional structure of the program under test. Second, it must recognize the statements which lead to executable object code and must arrange to record those statements actually executed when the program is run. If called for, clock readings and information about the program variables will also be recorded at this time. Third, it must encompass software for processing the primary data (execution counts) and other recorded data to yield the information desired by the investigator.

The first function will be implemented by a preprocessor. The tool will contain software which when applied to a program element or system compilable in CMS-2M, CMS-2Y, or CMS-2Q will produce a structural analysis of the program. The structural analysis will be manifested by the directed graph (or a suitable spanning subgraph thereof) representing the program under test in which each sequence (concatenation of one or more statements, occurring once each in order, with no transfers of control) of the source program is represented by an arc, and each decision point by a node. The graph need not be explicitly drawn as a geometric figure but may be given in an understandable equivalent notation.

The second function will also be carried out by the preprocessor. A set of options will be provided to the user of the validation tool for indicating the particular information he desires to receive in connection with the tests being run. Depending upon his choices, probes will be inserted into appropriate points of the original program and will be given suitable identification. Each probe consists of a transfer of control to a subroutine which counts the times the probe is activated as the program is being run, keeps a record of the count (and perhaps the contents of specified operational registers), and returns control to the program.

The input to the preprocessor will consist of the following items:

a. The instrumentation and analysis portions of the validation program.
b. The uninstrumented CMS-2 source text of the program to be examined.
c. The test data.
d. Control commands specifying the options desired by the user of the tool.

The output of the preprocessor will consist of the following items:

a. The source program listing with numbered statements and deletion of comments optional.
b. The structural characteristics of the source program with sufficient labeling to make all correspondences clear.
c. Identification of code segments and their coverage by the validation tool.
d. Identification of the probes and their locations.
e. Listing of input variables and their ranges.
f. The instrumented source code and test data constituting input to the compiler.

The printing of items b, c, d, or e just above may be suppressed at the option of the tester. Item f is not normally available for printout.

The output item f of the preprocessor will be compiled into an instrumented program, logically and functionally equivalent to the original program but with the probes added. The additional memory required by the instrumented program will not exceed 35% of the memory required by the original program. The real time required to run the instrumented program will be at most 35% greater than the time required to run the original program. The output of the compiler will be run on the appropriate computer and will comprise, as usual, results of the test run and, in particular:

a. The listing of test-case results in a format determined by the tester and
b. Error data, if any, detected by the compiler diagnostics.

The third function will be realized by a postprocessing program which analyzes and summarizes the data accumulated as a result of activation of the probes. It has no requirement for execution in real time. The tester may be supplied with any or all of the following items, provided he has requested them in advance:

a. Listing of all named procedures with the number of times each was entered.
b. Listing of procedures never entered.
c. Listing of all code segments, with the number of times each was executed.
d. The total number of direct code statements and the number executed.
e. Listing of declarative statements with the number of times each was called.
f. Listing of statements or code segments never exercised.
g. Listing of statements or code segments not instrumented.
h. Listing of all conditional transfers of control with the number of times each explicit alternative was taken.
i. Listing of all unconditional transfers of control.
j. Listing of variables declared and used.
k. Listing of global variables together with their use and type.
l. Listing of variables which change program control as well as where they are referenced.
m. Listing of parameters declared with their use and type.
n. Listing of input statements with their input data structures, showing (in the case of variables) the parts of the ranges that were tested and were not tested.
o. Intermediate values of designated input variables at specified points.
p. Listing of VARY operations with maximum and minimum values of loop indices and the number of times each loop was tested.
q. Clock readings at specified points.
r. Execution times for the program tests.
s. Cumulative statistics for the system as requested by the tester, comprising at most totals and means for those cases in which such parameters would have meaning.

3.1.1 ACCURACY AND VALIDITY

Since accurate mathematical computation is not an essential part of the validation tool, an accuracy requirement is not specified. Counts will be reported in decimal integers. Means, percentages, and other derived statistical data will be reported with two decimal

places to the right of the decimal point. Clock readings will be reported in terms of the accuracy and resolution provided by the computer being used.

3.1.2 TIMING

3.1.2.1 PREPROCESSING AND POSTPROCESSING PHASES. The timing will depend upon the information requested by the tester. The validation tool will be designed so that options available, but not selected by the tester, will be suppressed. Otherwise, the time requirement is only that imposed by reasonably efficient coding.

3.1.2.2 DATA COLLECTION PHASE. The running time of the instrumented program shall be at most 35% greater than the time to run the same program without instrumentation.

3.2 SYSTEM FUNCTIONS

The validation tool functions in three successive phases each time it is applied to a source program: preprocessing, data collection, and postprocessing. Note carefully that the terms "preprocessor" used in paragraph 3.1 and "preprocessing" used here are not precisely coextensive in their reference. The preprocessor is a program which governs both the preprocessing and the data collection phases.

3.2.1 PREPROCESSING

The purpose of the preprocessing phase is to convert a CMS-2 source program into an instrumented program ready to be compiled with test data to yield test results. Preprocessing takes place before the test program is run.

3.2.2 DATA COLLECTION

The purpose of the data collection phase is to monitor the execution of the tests in progress in order to capture data during the execution. The primary data will be counts of the executions of certain sequences. Whenever the sequence is executed, control will be first transferred to a subroutine which advances a counter identified with that sequence and then returned to the test program. If called for, clock readings and information about the program variables will be recorded at this time also. By its nature, data collection necessarily takes place while the test program is being run.

3.2.3 POSTPROCESSING

The purpose of the postprocessing phase is to process the results of the data collection phase, and report and summarize the data in a form specified by the tester based upon

options furnished by the tool. Postprocessing takes place after the test program has been run.

3.3 INPUTS/OUTPUTS

The CMS-2 source program and the test data will be on cards for input to a standard card reader for CMS-2 compilers. The CMS-2 monitor will also process input data from magnetic tape provided it is in card image format. The validation tool must, therefore, be listed in this format; if it is written in some language other than CMS-2, then the preprocessor must make the conversion.

Each card will contain a card identification field (columns 1 through 10) and a statement field (columns 11 through 80). The card identification field should contain the program identification (columns 1-4), a card sequence number (columns 5-8), and an insert number (columns 9-10). The statement field may utilize any format that serves the purpose of the validation tool. A card may contain more than one statement or a statement may occupy more than one card. The end-of-statement marker is a dollar sign.

Since the validation tool allows the tester to choose the content and the format of the output within a range of options, a selection algorithm for this purpose must be provided.

3.4 DATA CHARACTERISTICS

The data storage requirements for the preprocessing phase depend upon the length of the program under test and the volume of test data which will vary from application to application. The tool should be able to operate on source programs of up to 10 000 executable statements or to a total of 25 000 statements including comments. It should be able to label and store up to 100 000 paths through the graph of the source code with a binary indicator for each marking whether the path was executed or not. No data are to be permanently stored.

The data collection phase must be able to count and record the number of executions of each of up to 3 000 code segments. The extra storage required for the instrumented program and execution data must not exceed 35% of the storage used by the uninstrumented program.

The postprocessing phase must contain subroutines for implementing the options offered to the tester. No data are to be permanently stored by the validation tool; if the tester desires to maintain archives of test results, he does so independently. Consequently, there is no growth of data resulting from the use of the validation tool.

3.5 FAILURE CONTINGENCIES

The failure of the validation tool will have no direct effect on the performance of the program or system being tested. Hence, no back-up, fallback, or restart provisions need be made.

4.0 ENVIRONMENT

4.1 EQUIPMENT ENVIRONMENT

Since the validation tool is intended to be used for tests on software which will eventually become part of a Navy tactical system, they must be performed on appropriate equipment, the exact configuration of which is determined by the test environment.

a. The central processor will be the AN/UYK-7 for CMS-2Y and the AN/UYK-20 for CMS-2M. Approximately 1K of memory is required for the data collection phase of the tool program during execution of a test. Preprocessing and postprocessing phases can be run on any available central processor that can accept the implementation language of the tool.

b. Tape or disk storage is required for all three phases. The amount of storage required is fixed for any particular application of the tool but will be variable from application to application depending on the size and complexity of the software being tested, the number and nature of the tests, and the tool options chosen by the tester. A maximum of 100K cells is estimated for the total required.

c. The validation tool takes no special input or output device. Console input or output is not a requirement, although it could conceivably be a useful adjunct when certain procedures are used repetitively.

4.2 SUPPORT SOFTWARE ENVIRONMENT

The validation tool, whatever its implementation language, is intended to elicit information about the operation of software written for the principal tactical computers in use in the Navy. To be meaningful, the tests must be carried out in the presence of support software that reasonably approximates what will be encountered in actual operation.

4.3 INTERFACES

Data transfer as such does not take place between the validation tool and the software to which it is applied. The tool acts as an external observer, obtaining whatever information it needs through noninterfering probes.

4.4 SECURITY

Since the data used to exercise the software are test data and not data accumulated as the result of a real operation, security is not an issue.

5.0 COST FACTORS

5.1 NATURE OF THE COSTS

The requirements set forth herein are after the fact in the sense they superimpose a test tool on software that has been already designed and written. Since the test

tool is an item of capital equipment, its cost is onetime and outside the budget of any particular system. If properly constructed, the tool will be suitable for testing a wide variety of software.

The application of the tool to any piece of software is a matter for the judgment of the quality assurance investigator. It will depend, among other things, upon the complexity and importance of the particular program to be tested as well as the prevailing policy on the thoroughness of testing. For this reason the testing cost will vary greatly from instant to instant. How to amortize the initial cost of the tool and how to levy testing charges are questions outside the scope of these requirements.

5.2 CONSEQUENCES OF FURTHER DEVELOPMENT

If the testing is at all effective, it will have certain cost consequences. The results of the tests may well indicate that the subject software needs to be reworked before it can be certified, obviously entailing further costs. The effect of the tool on overall costs should be a net reduction due to the increased reliability of software that is certified with the use of the tool.

6.0 DEVELOPMENTAL PLAN

6.1 MANAGEMENT POLICY

Development and implementation of the proposed validation tool should be under the management of trained individuals in the Navy Department who can provide close technical monitoring. Watchfulness is essential because of the anticipated wide use of the tools once they are available. Many decisions will have to be made on the spot since the tools are envisioned collectively to be extremely flexible and this implies that their development must be a flexible process.

The tools themselves will have to be rigorously tested. Tests will be constructed to verify that the tool is capable of performing each of the functions listed in paragraph 3.1. Since the user of the tool is allowed a great deal of choice in the output he receives, special attention must be given to the subroutine that accepts and executes the user's selections to make sure that it responds to all (and only) the user's requests. If the validation tool is written in CMS-2, it may be used to monitor its own tests just as if it were a tactical program. The expected complexity of the tool will be much less than that of the tactical software it is designed to be applied to.